Digital Signal Processing, Mini-Project

Author: Ying Yiwen
Number: 12210159

Abstract

This paper presents a small project based on digital signal processing, aiming to realize the whole process from frequency generation to music production. Firstly, the notes are mapped to frequencies by mathematical formulas, and the frequency calculation is completed by combining the scale and octave adjustments; secondly, the sine function is used to generate the sound signals, and the envelope decay function is used to simulate the vibration effect of the instruments, while exploring the influence of the harmonic components on the timbre; lastly, through the programmed control of the parameters of the notes, rhythms, scales, etc., we have successfully generated complete the music works.

Contents

1 Generate Frequency

1.1 Principle

For note to frequency conversion we need to use a dictionary to convert 7 notes to 12 intervals, and for adding and subtracting octaves the high and low notes are converted to changes in frequency. We also need to add a bias to the gain based on the key signature (e.g. 1=C). After getting the bias, you can use the formula(1) to get the frequency of that particular note.

$$freq = key * 2^{\frac{bias}{12}} \tag{1}$$

$$bias = interval + note * 12 \tag{2}$$

1.2 Code Implementation

表 1. C 调中音符与频率(取整后)对应表，单位为 Hz

	1（C）	2(D)	3(E)	4(F)	5(G)	6(A)	7(B)
低音	262	294	330	349	392	440	494
中音	523	587	659	698	784	880	988
高音	1046	1175	1318	1397	1568	1760	1976
相邻音符频率比 $n+1/n$	1.12	1.12	1.057	1.12	1.12	1.12	1.058

(a) Table of notes and intervals in C major

表 2 不同调号主音频率，单位 Hz

C 调频率	261.5	293.5	329.5	349	391.5	440	494
C 调音符(英文名)	1(C)	2(D)	3(E)	4(F)	5(G)	6(A)	7(B)
C 调	1=C (261.5)						
D 调		1=D (293.5)					
E 调			1=E (329.5)				
F 调				1=F (349)			
G 调					1=G (391.5)		
A 调						1=A (440)	
B 调							1=B (494)

(b) Table of Master Tone Frequencies

Fig. 1: Transfer Table

We can test some specific values.

2.1 Tune

```
% under 1A
tone2freq(1,0,0) % A
```

```
ans = 440
```

```
tone2freq(3,1,0) % C+8
```

```
ans = 1.0465e+03
```

```
tone2freq(4,0,1) % D#
```

```
ans = 622.2540
```

Fig. 2: Test1

According to the table, we can write the code as bellow:

```
function freq = tone2freq(tone, scale, noctave, rising)
  % tone: 1-7
  % scale: 'A', 'B', 'C', 'D', 'E', 'F', 'G'
  % noctave: octave offset, +-8
  % rising: pitch adjustment, +-1
  % freq: output frequency

```

```
    scale_offsets = containers.Map({'A', 'B', 'C', 'D', 'E', 'F', 'G'}, [0, 1, -5,
        -4, -3, -2, -1]);
    offset = scale_offsets(scale);
    new_tone = mod(tone + offset - 1, 7) + 1; % 1-7
    octave_adjust = floor((tone + offset - 1) / 7);
    dic = [0, 2, 3, 5, 7, 8, 10];
    factor = (dic(new_tone) + (noctave + octave_adjust) * 12 + rising) / 12;
    freq = 440 * 2^factor;
end
```

2.2 Scaled

```
tone2scaledfreq(1,'B',0,0) % under 1B, B
```

```
ans = 493.8833
```

```
tone2scaledfreq(3,'C',1,0) % under 1C, E
```

```
ans = 659.2551
```

```
tone2scaledfreq(6,'F',0,-1) % under 1F, Cb
```

```
ans = 554.3653
```

Fig. 3: Test2

```
for i = 1:15
    tone2scaledfreq(i,'A',0,0)
end
```

```
ans = 440
ans = 493.8833
ans = 523.2511
ans = 587.3295
ans = 659.2551
ans = 698.4565
ans = 783.9909
ans = 880
ans = 987.7666
ans = 1.0465e+03
ans = 1.1747e+03
ans = 1.3185e+03
ans = 1.3969e+03
ans = 1.5680e+03
ans = 1760
```

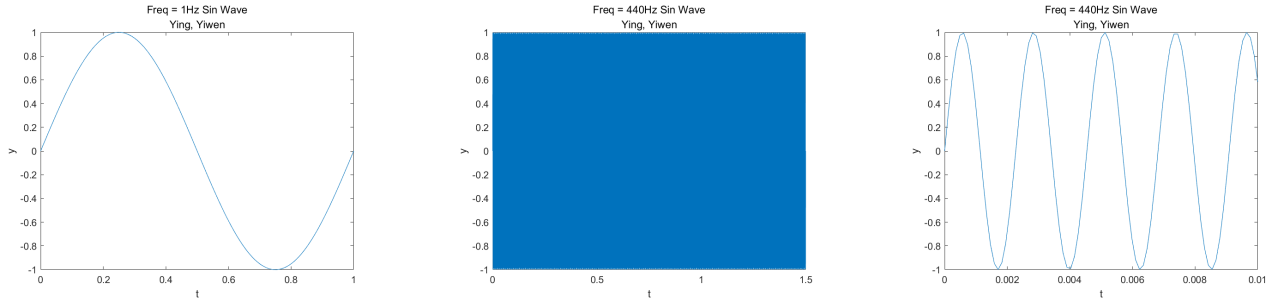
Fig. 4: Test3

2 Generate Sound

2.1 Basic Sound

As usual, we use sine function to generate the music sound. It looks like fig.2(a).

```
fs = 8192; f = 1; T = 1/f; t = linspace(0, T, fs);
y = sin(2*pi*f*t);
plot(t,y),xlabel('t'),ylabel('y'),title('Freq = 1Hz Sin Wave','Ying, Yiwen');
sound(y,fs); pause(length(y)/fs);
```



(a) Sine Function (b) Sine Function(Tune) (c) Enlarged View of Tune

Fig. 5: Signal Plot

For the fig2.(a), it's frequency is 1 Hz, which can't be heard by human ear. We adjust the frequency to be 440 Hz, and then we can here 'du'. The signal is like fig2.(b), and it's detail is in fig2.(c).

With frequency and duration known, we can use the above way to generate sound.

```
function waves = gen_wave(tone, scale, noctave, rising, rhythm, fs)
  % tone: 1-7
  % scale: 'A', 'B', 'C', 'D', 'E', 'F', 'G'
  % noctave: octave offset, +-8
  % rising: pitch adjustment, +-1
  % rhythm: time, 1 for normal
  % fs: sample rate

  if tone == 0
    t = linspace(0, rhythm, fs*rhythm);
    waves = zeros(size(t));
  elseif (1 <= tone) && (tone <= 7)
    f = tone2freq(tone, scale, noctave, rising);
    t = linspace(0, rhythm, fs*rhythm);
    waves = 0.7*sin(2*pi*f*t)+0.2*sin(2*pi*2*f*t)+0.05*sin(2*pi*3*f*t)+0.05*sin(2*
        pi*4*f*t);
    waves = waves .* max(0, (1-0.25*t/rhythm).^2);
  end
end
```

2.2 Attenuation

Considering that when a musical instrument is played, the vibrations decay and do not continue to vibrate at a fixed amplitude, an envelope decay function can more realistically simulate the music production. We test kinds of attenuation function, like exponential, linear, squared, oscillating.

```
fs = 8192; f = 440; rhythm = 1.5;
t = linspace(0, rhythm, fs*rhythm);
y = sin(2*pi*f*t);
waves1 = y .* exp(-t/rhythm); % exponential
waves2 = y .* max(0, 1-0.5*t); % linear
waves3 = y .* max(0, (1-0.5*t).^2); % squared
waves4 = y .* exp(-t/rhythm) .* sin(2*pi*50*t); % oscillating
```

Squared attenuated have the best performance. The third wave is very much like strings, and the fourth wave vibrates like a metal instrument.

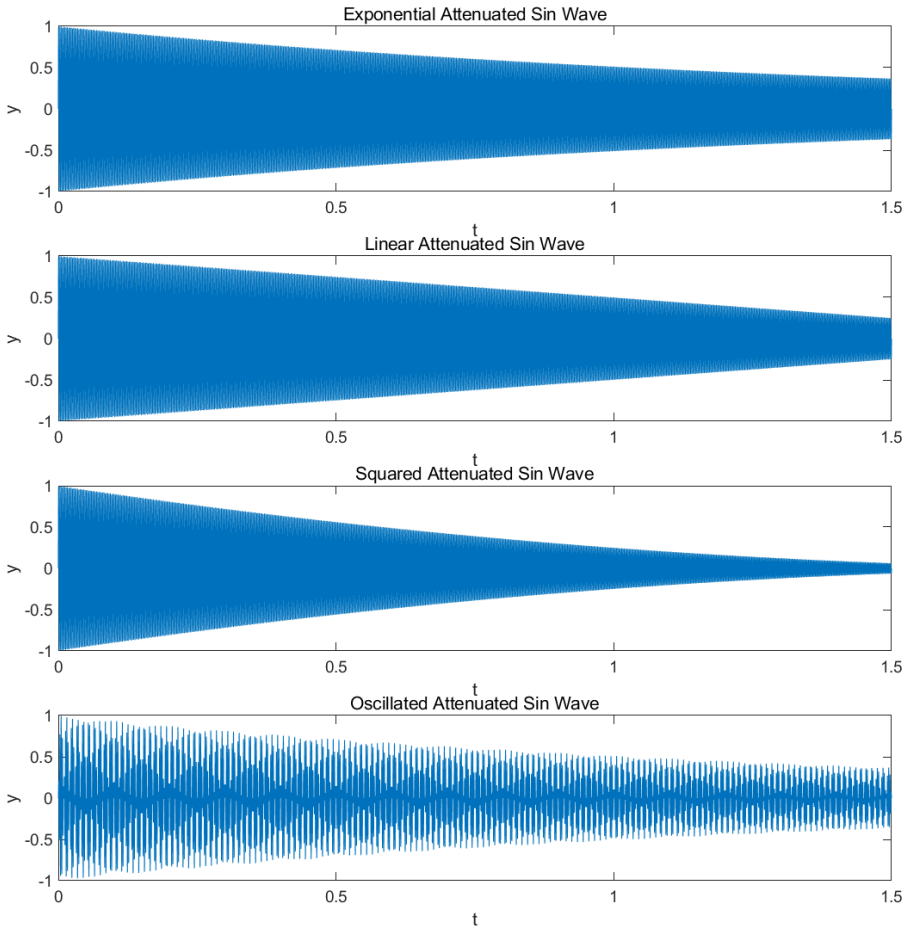


Fig. 6: Different Attenuation

2.3 Harmonic

When music is played on a musical instrument, in addition to the fundamental frequency sound in the musical score, a variable number of standing waves are generated due to the sound generating principle of the instrument. Standing waves mean that when a string is fixed at both ends, and we pluck the string, the length of the vibrating part of the string must be an integer multiple of the half-wavelength, i.e., the frequency of the sound emitted consists of the fundamental frequency as well as integer multiples of the harmonics of the fundamental frequency. We test different weight of each harmonics of the fundamental frequency.

```
fs = 8192; f = 220; rhythm = 1;
t = linspace(0, rhythm, fs*rhythm);
wave1 = 0.8*sin(2*pi*f*t)+0.1*sin(2*pi*2*f*t)+0.05*sin(2*pi*3*f*t)+0.05*sin(2*pi*4*f*t);
wave2 = 0.7*sin(2*pi*f*t)+0.2*sin(2*pi*2*f*t)+0.05*sin(2*pi*3*f*t)+0.05*sin(2*pi*4*f*t);
wave3 = 0.6*sin(2*pi*f*t)+0.2*sin(2*pi*2*f*t)+0.1*sin(2*pi*3*f*t)+0.1*sin(2*pi*4*f*t);
wave4 = 0.5*sin(2*pi*f*t)+0.25*sin(2*pi*2*f*t)+0.15*sin(2*pi*3*f*t)+0.1*sin(2*pi*4*f*t);
```

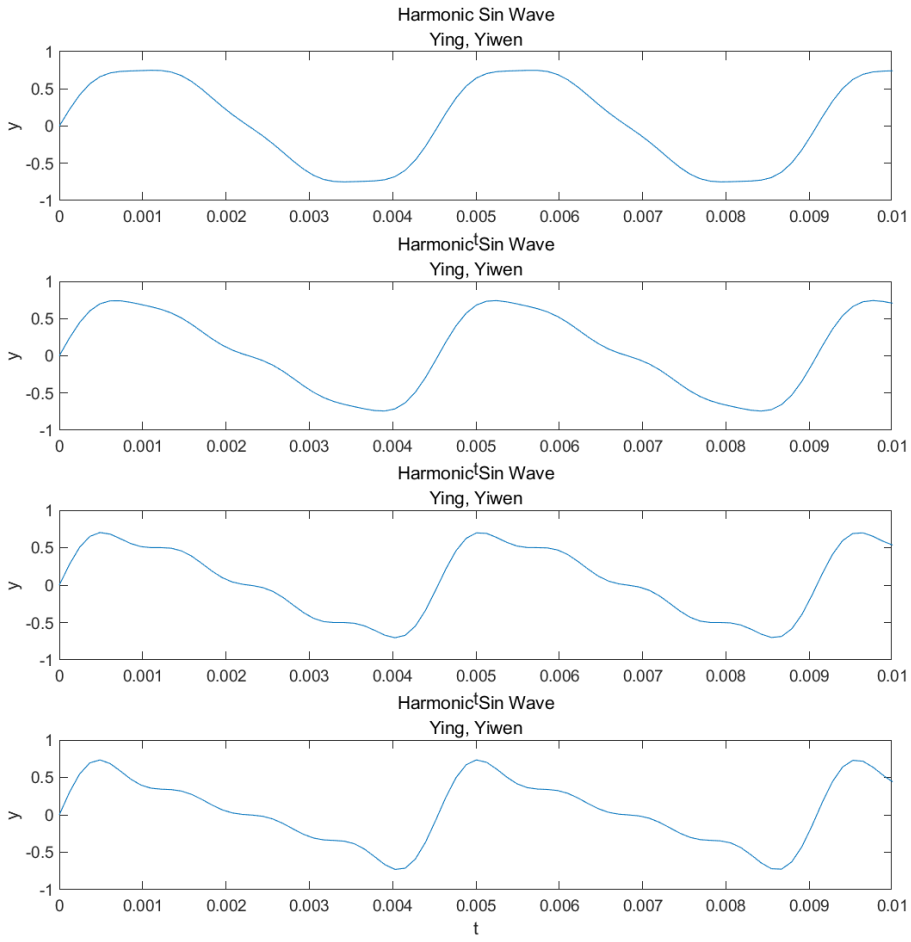


Fig. 7: Different Harmonic Component

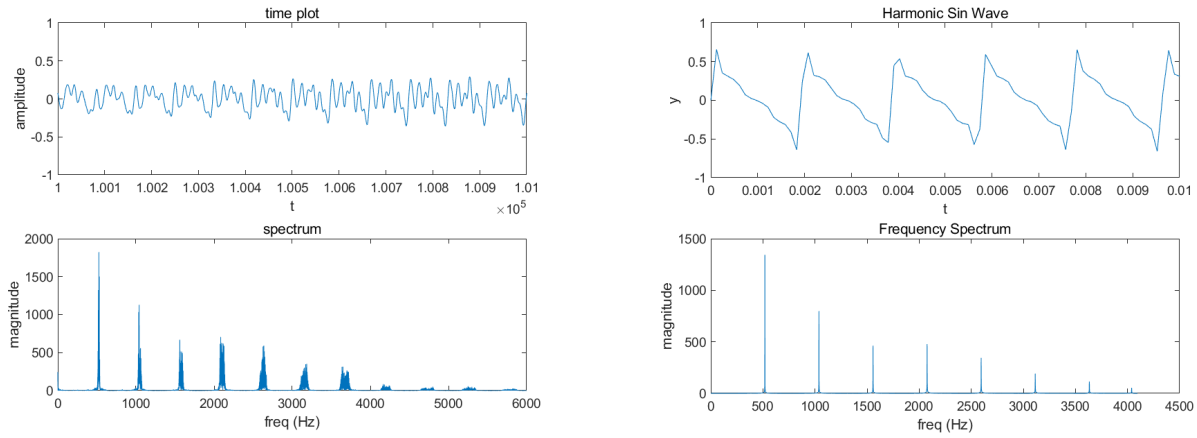
Judging by listening, I think the second way has the best sound.

2.4 Simulate Violin

We can analyze the true sound of violin, and simulate its frequency spectrum, to get the best immitation of it.

```
fs = 8192; f = 519; rhythm = 1;
t = linspace(0, rhythm, fs*rhythm);
wave = 0.33*sin(2*pi*f*t)+0.20*sin(2*pi*2*f*t)+0.12*sin(2*pi*3*f*t)+0.13*sin(2*pi
    *4*f*t)+0.10*sin(2*pi*5*f*t)+0.06*sin(2*pi*6*f*t)+0.04*sin(2*pi*7*f*t)+0.02*sin
    (2*pi*8*f*t);
figure;
subplot(2,1,1),plot(t,wave),xlabel('t'),ylabel('y'),title('Harmonic Sin Wave'),
    axis([0 0.01 -1 1]);
n = length(wave);
frequencies = (0:n-1)*(fs/n);
```

```
wave_fft = abs(fft(wave));
subplot(2,1,2),plot(frequencies(1:floor(n/2)), wave_fft(1:floor(n/2))),xlabel('
    freq (Hz)'),ylabel('magnitude'),title('Frequency Spectrum');
saveas(gcf,'violin_mine.png');
sound(wave,fs);
pause(length(wave)/fs);
```



(a) True Violin

(b) Generated by Code

Fig. 8: Spectrum Anlayzation

Similarly, we can simulate the envelope of the violin. We use short-term energy method to get the envelope of the violin, and use cubic spline interpolation to scale it to the rhythm, then apply it on the signal.

```
[audioData, fs_input] = audioread('violin.mp3');
start_sample = floor(1 * fs_input);
end_sample = floor(3.5 * fs_input);
audioData = audioData(start_sample:end_sample);

% Short-term energy calculations
frame_length = 8192; hop_length = 2048; window = hamming(frame_length);
num_frames = floor((length(audioData) - frame_length) / hop_length) + 1;
energy = zeros(1, num_frames);
for i = 1:num_frames
start_idx = (i-1)*hop_length + 1;
end_idx = start_idx + frame_length - 1;
frame = audioData(start_idx:end_idx) .* window;
energy(i) = sum(frame.^2);
end

% Target audio signal
fs = 8192; rhythm = 1; f = 525;
t = linspace(0, rhythm, fs * rhythm);
wave = 0.33*sin(2*pi*f*t) + 0.20*sin(2*pi*2*f*t) + 0.12*sin(2*pi*3*f*t) + ...
0.13*sin(2*pi*4*f*t) + 0.10*sin(2*pi*5*f*t) + 0.06*sin(2*pi*6*f*t) + ...
0.04*sin(2*pi*7*f*t) + 0.02*sin(2*pi*8*f*t);

% Normalize and adjust the envelope to the target audio length
time_energy = linspace(0, length(audioData)/2.5*rhythm/fs_input, num_frames);
```

```
time_wave = linspace(0, rhythm, length(wave));
envelope = interp1(time_energy, energy, time_wave, 'spline');
envelope = envelope / max(envelope);

modulated_wave = wave .* envelope;

figure;
subplot(3, 1, 1), plot(linspace(0, length(audioData)/fs_input, length(audioData)),
    audioData), title('Input Audio');
subplot(3, 1, 2), plot(time_wave, envelope), title('Envelope (Scaled to Target)');
subplot(3, 1, 3), plot(time_wave, modulated_wave), title('Modulated Audio');
saveas(gcf, 'violin_modulated.png');
sound(modulated_wave, fs);
pause(length(modulated_wave)/fs);
audiowrite('modulated_audio.wav', modulated_wave, fs);
```

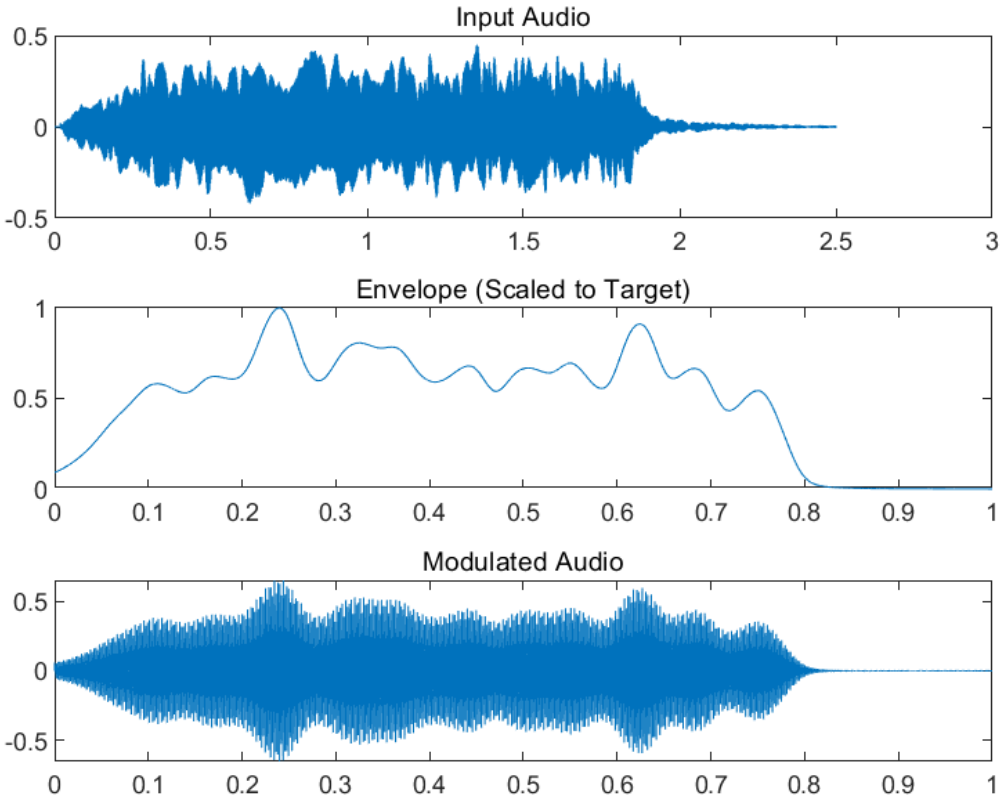


Fig. 9: Envelope and Frequency Modulated Signal

3 Generate Music

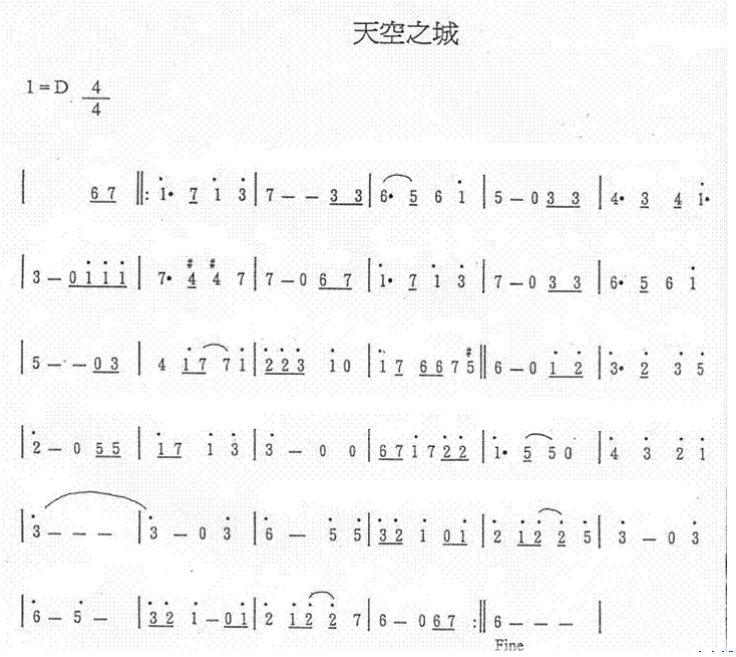
3.1 Code Implementation

Being able to generate a single tune, we can also generate music.

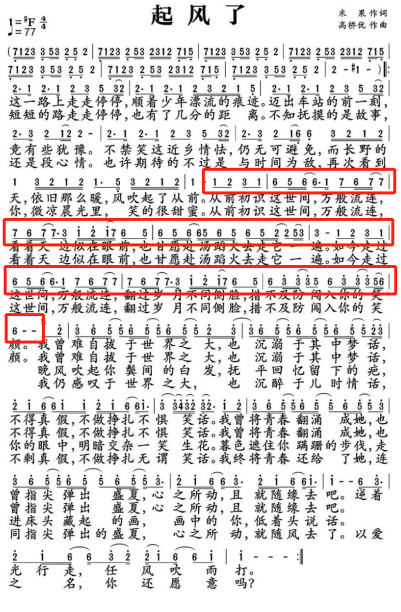
```
function waves = gen_music(tones, scale, noctaves, risings, rhythms, fs)
  % tones: 1-7, list
  % scale: 'A', 'B', 'C', 'D', 'E', 'F', 'G'
  % noctaves: octave offset, +-8, list
  % risings: pitch adjustment, +-1, list
  % rhythms: time, 1 for normal, list
  % fs: sample rate

  waves = [];
  for i = 1:length(tones)
    new_wave = gen_wave(tones(i), scale, noctaves(i), risings(i), rhythms(i), fs);
    waves = [waves, new_wave];
  end
end
```

3.2 Example



(a) Castle in the Sky



(b) The Wind Rises

Fig. 10: Music Score

We can use such code to get the music "Castle in the Sky":

```
% set parameter
tones = [6,7, 1,7,1,3, 7,3,3, 6,5,6,1, 5,0,3,3, 4,3,4,1,...
  3,0,1,1,1, 7,4,4,7, 7,0,6,7, 1,7,1,3, 7,0,3,3, 6,5,6,1,...
  5,0,3, 4,1,7,7,1, 2,2,3,1,0, 1,7,6,6,7,5, 6,0,1,2, 3,2,3,5,...
  2,0,5,5, 1,7,1,3, 3,0,0, 6,7,1,7,2,2, 1,5,5,0, 4,3,2,1,...
  3, 3,0,3, 6,5,5, 3,2,1,0,1, 2,1,2,2,5, 3,0,3,...
  6,5, 3,2,1,0,1, 2,1,2,2,7, 6,0,6,7, 6];
scales = 'D';
```

```
noctaves = [0,0, 1,0,1,1, 0,0,0, 0,0,0,1, 0,0,0,0, 0,0,0,1,...
  0,0,1,1,1, 0,0,0,0, 0,0,0,0, 1,0,1,1, 0,0,0,0, 0,0,0,1,...
  0,0,0, 0,1,0,0,1, 1,1,1,1,0, 1,0,0,0,0,0, 0,0,1,1, 1,1,1,1,...
  1,0,0,0, 1,0,1,1, 1,0,0, 0,0,1,0,1,1, 1,0,0,0, 1,1,1,1,...
  1, 1,0,1, 1,1,1, 1,1,1,0,1, 1,1,1,1,1, 1,0,1,...
  1,1, 1,1,1,0,1, 1,1,1,1,0, 0,0,0,0, 0];
risings = [0,0, 0,0,0,0, 0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,...
  0,0,0,0,0, 0,1,1,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,...
  0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0,1, 0,0,0,0, 0,0,0,0,...
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,0,0, 0,0,0,0, 0,0,0,0,...
  0, 0,0,0, 0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,...
  0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0, 0];
rhythms = [0.5,0.5, 1.5,0.5,1,1, 3,0.5,0.5, 1.5,0.5,1,1, 2,1,0.5,0.5,
    1.5,0.5,0.5,1.5,...
  2,0.5,0.5,0.5,0.5, 1.5,0.5,1,1, 2,1,0.5,0.5, 1.5,0.5,1,1, 2,1,0.5,0.5,
      1.5,0.5,1,1,...
  3,0.5,0.5, 1,0.5,0.5,1,1, 0.5,0.5,0.5,1,1, 1,0.5,0.5,0.5,1,1, 2,1,0.5,0.5,
      1.5,0.5,1,1,...
  2,1,0.5,0.5, 0.5,0.5,1,1, 2,1,1, 0.5,0.5,1,1,0.5,0.5, 1.5,0.5,1,1, 1,1,1,1,...
  4, 2,1,1, 2,1,1, 0.5,0.5,1,0.5,0.5, 1,0.5,0.5,0.5,1, 2,1,1,...
  2,2, 0.5,0.5,2,0.5,0.5, 1,0.5,0.5,0.5,1, 2,1,0.5,0.5, 4];
fs = 44100;

% generate music and save
music_wave = gen_music(tones, scales, noctaves, risings, rhythms, fs);
sound(music_wave, fs);
audiowrite('generated_music.wav', music_wave, fs);
disp('Saved as generated_music.wav');
```

Similarly, I choose "The Wind Rises", because the key is #F, we set scales to be 'F', and all risings to be 1:

```
% set parameter
tones = [1,2,3,1,6,5,6,6,1,7,6,7,7,...
  7,6,7,7,3,1,2,1,7,6,5, 6,5,6,6,5,6,5,6,5,2,2,5,3, 3,1,2,3,1,...
  6,5,6,6,1,7,6,7,7, 7,6,7,7,3,1,2,1,7,6,5, 6,3,3,3,5,6,3,3,3,5,6,6];
scales = 'F';
noctaves = [zeros(1,13),...
  0,0,0,0,0,1,1,1,0,0,0, zeros(1,13), 0,0,0,0,0,...
  zeros(1,9), zeros(1,5), 1,1,1,0,0,0, 0,1,1,1,0,0,1,1,1,0,0,0];
risings = ones(1,74);
rhythms = [0.5,0.5,0.5,0.5,0.5,0.25,0.25,0.75,0.25,0.5,0.25,0.25,1,...
  0.5,0.25,0.25,0.75,0.25,0.25,0.25,0.25,0.25,0.5,0.5,
      0.5,0.25,0.25,0.25,0.25,0.25,0.25,0.5,0.25,0.25,0.5,0.25,0.25,
      2,0.5,0.5,0.5,0.5,...
  0.5,0.25,0.25,0.75,0.25,0.5,0.25,0.25,1,
      0.5,0.25,0.25,0.75,0.25,0.25,0.25,0.25,0.25,0.5,0.5,
      0.5,0.25,0.25,0.5,0.5,0.5,0.25,0.25,0.5,0.25,0.25,3];
fs = 44100;

% generate music and save
music_wave = gen_music(tones, scales, noctaves, risings, rhythms, fs);
sound(music_wave, fs);
audiowrite('generated_music_mine.wav', music_wave, fs);
disp('Saved as generated_music_mine.wav');
```

4 Conclusion

This project successfully demonstrates the application of digital signal processing to audio synthesis and music generation. By converting notes to frequencies, sound signals with realistic decay effects are generated and these elements are combined into complete compositions. The results demonstrate the ability to synthesize high-quality sounds and replicate the characteristics of various instruments. Envelope decays and harmonic combinations were repeatedly adjusted in the process of realizing musical synthesis. I realized that the difference in timbre between different instruments is actually controlled by the frequency harmonic components and the way they are attenuated, and I also understood how to synthesize realistic music through code. At the same time, in the process of writing functions to convert from simple scores to music, I also understood the knowledge of music theory, and deeply felt that it is really an interesting thing to realize music synthesis by code!